1. With the expression described in the problem statement

```
if (e() || f() && g() || h())
do_something();
if (e() && f() || g() && h())
do_something();
if (e() && (f() || g()))
do_something();
```

The first line is constructed where if the first set of or operands e() and f() return a false value, then the next set of or operands on the other side of the && operator will not be executed. Thus, do_something() will not be executed as well. The second line is constructed where if either e() or (f() || g()) returns false, then the statement will short circuit and the remaining expressions of the && statements will not be executed. Thus, do_something() will also not be executed. The third line is constructed where if e() returns false, then the statement will short circuit and then the (f() || g()) will not return false. Thus, in that case do_something() will not be executed.

2.

```
class Node:
    def __init__(self, val):
        self.value = val
        self.next = None
class HashTable:
    def __init__(self, buckets):
        self.array = [None] * buckets
def insert(self, val):
    bucket = hash(val) % len(self.array)
    tmp_head = Node(val)
    tmp_head.next = self.array[bucket]
    self.array[bucket] = tmp_head
```

a.

```
# generator function, this would be integrated into the hashtable class
    def __iter__(self):
        for bucket in self.array:
            node = bucket
            while node is not None:
                yield node.value
                node = node.next
```

b.

```
# Python generator function capable of iterating over all of the items in your
HashTable container
# This goes inside the hashtable class
    def __iter__(self):
        it = NewIterator(self.array)
        return it
```

```
class NewIterator:
    def __init__(self, arr):
        self.arr = arr
        self.pos = 0
    # inside the NewIterator Class
    def __next__(self):
        while self.bucket_index < len(self.hash_table.array):
            if self.node is None:
                self.node = self.hash_table.array[self.bucket_index]
            if self.node is not None:
                value = self.node.value
                self.node = self.node.next
                return value
            self.bucket_index += 1
            self.node = None
        raise StopIteration
```

c.

```
for value in hash_table:
    print(value)
```

d.

```
ht = HashTable(10)
# ... (insert some values into the hash table)
iterator = ht.__iter__()
while True:
    try:
        value = iterator.__next__()
        print(value)
    except StopIteration:
        break
```

e.

```
"""In this case, the f in the forEach method would be the lambda input, we would put
the forEach method in the HashTable class"""
 def forEach(self, f):
    for x in self.array:
      f(x)
```

3. a.

```
X -> green
```

b.

```
false
```

c.

```
Q -> tomato
Q -> beet
```

d.

```
Q -> celery, R -> green
Q -> tomato, R -> red
Q -> persimmon, R -> orange
Q -> beet, R -> red
Q -> lettuce, R -> green
```

4. a.

```
% Rules:
likes_red(X, Y) :- food(Y), color(Y, red), likes(X, Y)
likes_red(X, Y) :- likes(X, Y), color(Y, red), food(Y)
```

b.

```
% Rules:
likes_foods_of_colors_that_menachen_likes() :-
likes_foods_of_colors_that_menachen_likes(X) :-
    likes(menachen, FoodMenachenLikes),
    color(FoodMenachenLikes, Color),
    likes(X, FoodXLikes),
    color(FoodXLikes, Color), X \= menachen
```

5.

```
% Rule:
reachable(X, Y) :- road_between(X, Y).
reachable(X, Y) :- road_between(X, Z), reachable(Z, Y).
```

6. ``` foo(bar,bletch) with foo(X,bletch)

- unifies X = bar

foo(bar,bletch) with foo(bar,bletch,barf)

- does not unify because the arities of the two functors are different

foo(Z,bletch) with foo(X,bletch)

- Unifies X = Z

foo(X, bletch) with foo(barf, Y)

- Unifies barf = X, Y = bletch

foo(Z,bletch) with foo(X,barf)

- Unifies Z = X, bletch = barf

foo(bar,bletch(barf,bar)) with foo(X,bletch(Y,X))

- Unifies X = bar, Y = barf

foo(barf, Y) with foo(barf, bar(a,Z))

- Unifies barf = barf, Y = bar(a, Z)

foo(Z,[Z|Tail]) with foo(barf,[bletch, barf])

- Does not unify because bletch and Z are different from each other, since Z was already unified with barf

foo(Q) with foo([A,B|C])

- Unifies Q = [A, B | C]

foo(X,X,X) with foo(a,a,[a])

- Does not unify since X can't be both a value and a list of those values X != [a]

7.

```
% adds a new value X to an empty list
insert_lex(X,[],[X]).
% the new value is < all values in list
insert_lex(X,[Y|T],[X,Y|T]) :- X =< Y.
% adds somewhere in middle
insert_lex(X,[Y|T],[Y|NT]) :-
X > Y, insert_lex(X,T,NT).
```

8.

```
% count_elem(List, Accumulator, Total)
% Accumulator must always start at zero
count_elem([], Acc, Acc).
count_elem([_|Tail], Sum, Total) :-
 Sum1 is Sum + 1,
 count_elem(Tail, Sum1, Total).
```

9.

```
% fact
gen_list(_, 0, []).
% rule
gen_list(Value, N, [Value|Tail]) :-
 N > 0,
 N1 is N - 1,
 gen_list(Value, N1, Tail).
```

10.

```
append_item([], Item, [Item]).
append_item([Head|Tail], Item, [Head|ResultTail]) :-
    append_item(Tail, Item, ResultTail).
```